RMIT UNIVERSITY Computer Science and Information Technology

RMIT School of Computer Science and Information Technology

Programming 2

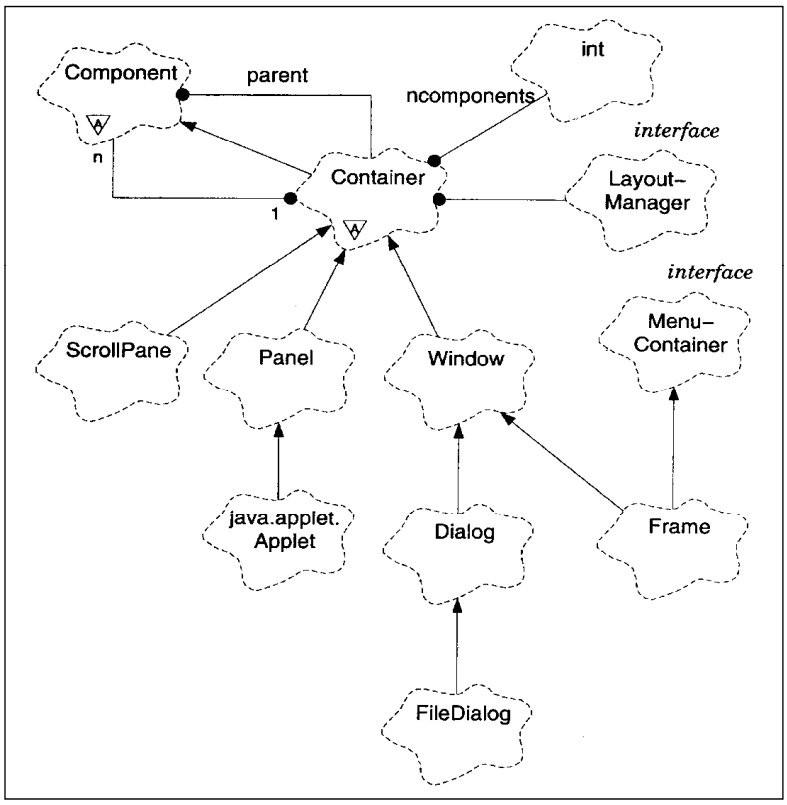
Topic 6: User Interfaces and Graphics II
Layout Management, Windows and Menus

Lecture Slides

Containers, Components & Layout Management

- The relationship between containers, their contained components and their layout manager is fundamental to the AWT (and Swing)
- A container is an AWT component that can contain other components (added using the `Container.add(Component)` method)
- The abstract class `java.awt.Container` extends `Component` and thus Containers may also contain other Containers (composite pattern)
- Containers group related components and treat them as a unit, which helps arrange components on the display
- Many AWT components and all Swing components subclass container (e.g. (J)Frame, (J)Applet, (J)Panel etc.)

Class Diagram – Component / Container Relationships



Layout Managers

- Containers are not responsible for laying out (sizing, positioning etc.) their own components => all containers have an instance of a layout manager
- A Layout Manager is a class that implements the `java.awt.LayoutManager` or `java.awt.LayoutManager2` interface
  - the LayoutManager2 interface allows a component to provide constraints that assist with resizing
  - uses overloaded `Container.add(Component comp, Object constraints)` method
- AWT provides 5 pre-defined layout managers
  - `FlowLayout*, BorderLayout*, GridLayout*, CardLayout` and `GridbagLayout` (* covered in this lecture)
  - Swing adds the `BoxLayout` and `SpringLayout` (low level for use with GUI Builders)
- Call Container's `void setLayout(LayoutManager mgr)` method to set a containers layout manager

Why use Layout Managers?

- Layout managers are used in Java in preference to fixed x,y,width,height positioning:
  - fixed positioning can be achieved by setting the layout manager to ‘null’
  - poor solution for windows that need to be resized because of font metrics, placement etc. => must be manually calculated (difficult!)
  - cross platform programming is tedious (because of differences in available fonts, component sizes etc.)
  - layout managers provide solutions for common layout scenarios (i.e. reuse instead of coding from scratch)

Component Preferred/Minimum Sizes

- components can specify *preferred* and *minimum* sizes that are used by the layout manager
- *default values automatically set by the component (or peer)
- can be changed in AWT but requires subclassing in order to override two Container methods:
  - `public Dimension getPreferredSize()`
  - `public Dimension getMinimumSize()`
  - Swing provides setter methods
- not all layout managers always abide by the components preferred and minimum sizes
- preferred and minimum sizes are usually set by the peer so generally don’t need to be set manually
- there is also a *maximum* size but is not used by standard AWT layout managers

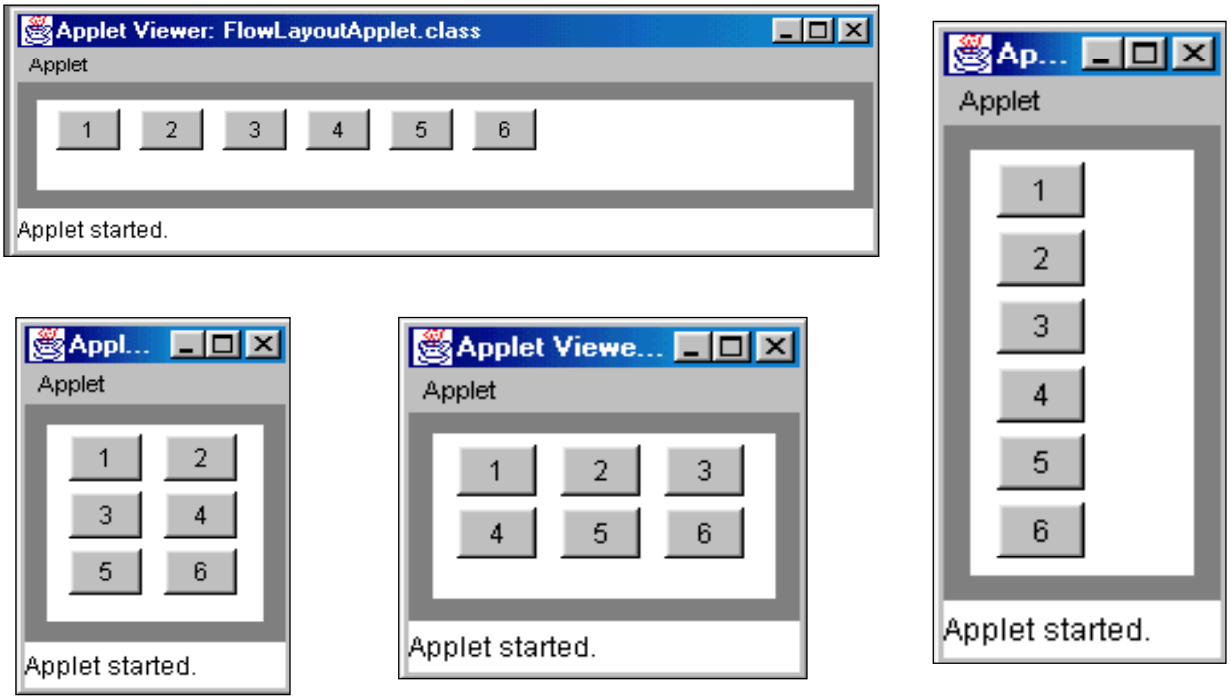
Flow Layout

- displays components left to right, top to bottom.
- default layout manager for panels and applets
- respects preferred size and height if component has not been explicitly sized with `setSize()`
- can set the spacing between components during construction or with `setHgap(int hgap)` or `setVgap(int vgap)`
- can set the alignment of components during construction or with `setAlignment(int align)`
  - `FlowLayout.CENTER, FlowLayout.LEFT, FlowLayout.RIGHT`

Contructors

- `FlowLayout()`
- `FlowLayout(int align)`
- `FlowLayout(int align, int hgap, int vgap)`

Flow Layout Resizing



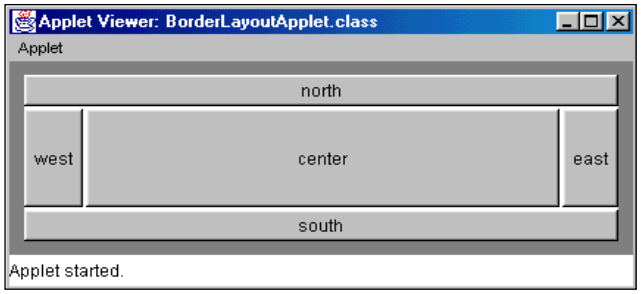
Border Layout

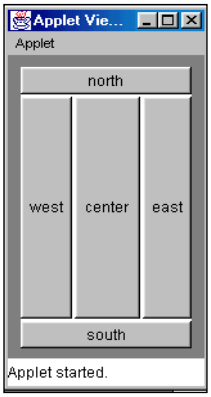
- Lays out at most five components into geometric regions
- Implements `LayoutManager2` interface so requires constraints which are string constants :
  - `BorderLayout.NORTH , BorderLayout.SOUTH , BorderLayout.WEST, BorderLayout.EAST & BorderLayout.CENTER`
- *north* and *south* components are stretched horizontally but preferred height is respected
- *west* and *east* components are stretched vertically but preferred width is respected
- *center* layout gets whatever space is left (preferred size is ignored)
- vertical and horizontal gap set in same manner as FlowLayout

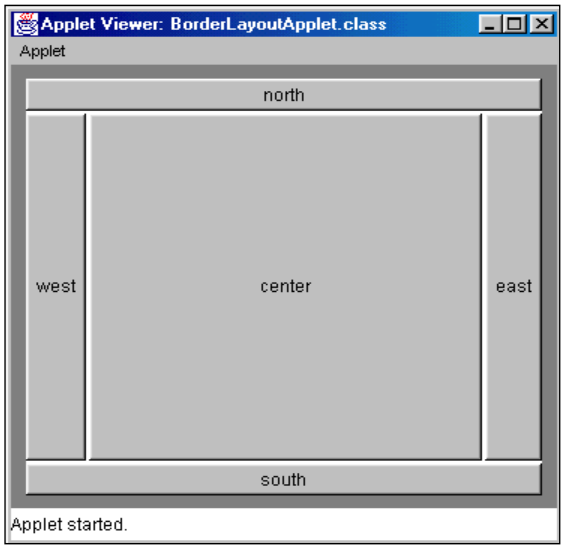
Constructors

- `BorderLayout()`
- `BorderLayout(int hgap, int vgap)`

Border Layout Resizing



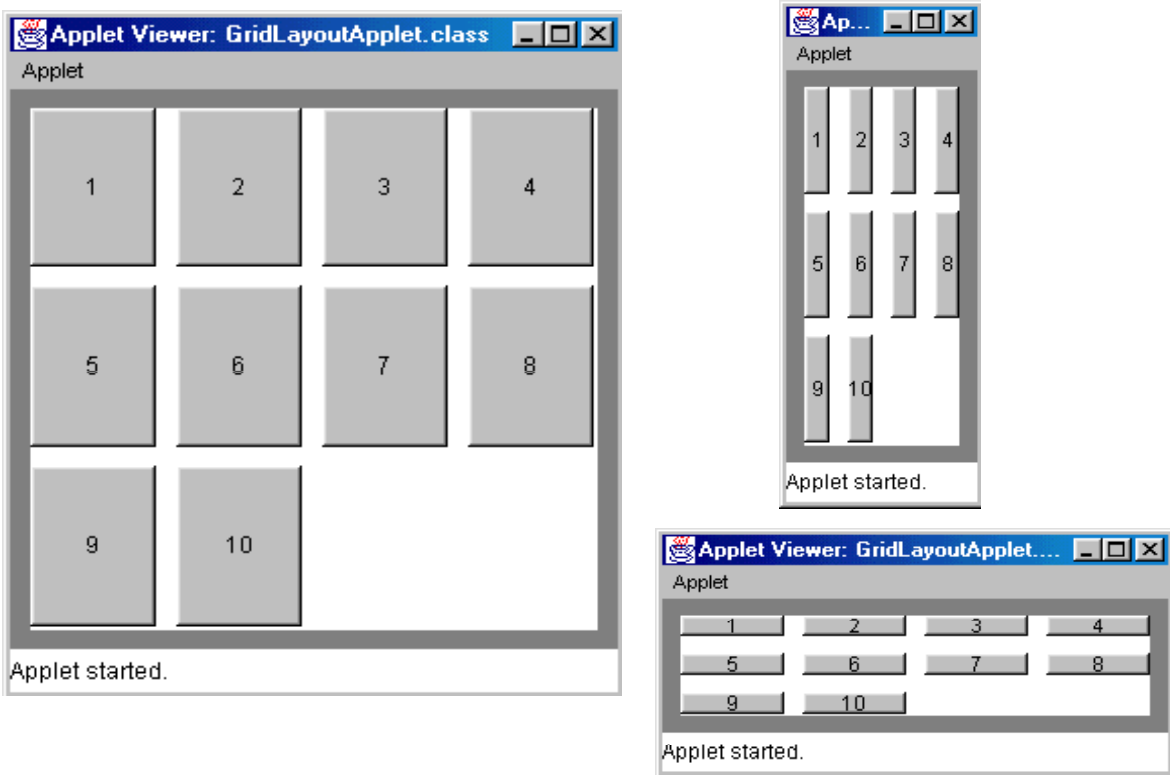




Grid Layout

- Lays out components in a grid
- Implements `LayoutManager` interface (i.e. there are no constraints)
- Number of cells are specified during construction
  - if a row or column is specified as 0 then size is automatically computed depending upon number of contained components
  - row and column cannot both be zero (throws exception)
- All cells are the same size (components grow or shrink to fill the display area)
  => preferred and *minimum* sizes of contained components are ignored
- Can set vertical gaps and horizontal gaps as with `BorderLayout` and `FlowLayout`
- Constructors:
  - `GridLayout()`
  - `GridLayout(int rows, int cols)`
  - `GridLayout(int rows, int cols, int hgap, int vgap)`

Grid Layout



GridBagLayout

- Most flexible but most complicated* of the layout managers
- Implements `LayoutManager2` interface
  - constraint is an instance of `GridbagConstraints` class
- Lays out components in a grid
  - components may span more than one cell
  - number of cells is not specified at construction but rather determined by the constraints which are set for each component
  - size of cells is not specified but is determined in relation to the size of the component it contains
- Particularly useful for laying out input forms
- Constructor
  - `GridBagLayout()`
- Not covered in Programming 2

Windows

- `java.awt.Window` provides a superclass with common functionality that is extended by both `Frame` and `Dialog`
- it subclasses `Container` allowing components to be added with the `add()` method
- generally not instantiated directly
- provides a number of general purpose window methods
  - inherited by `Frame` and `Dialog`
  - recall `WindowListener` and `WindowEvent` classes that are used with Windows

java.awt.Frame

- `java.awt.Frame` is a subclass of `java.awt.Window`
  - provides a border, title and optional menubar
  - may be resizable (depends on implementation)
  - can be minimized to an icon (depends on implementation)
  - can have an associated icon image: `setIconImage(Image image)`
- constructor: `Frame(String Title)`
  - *Title* represents the text to be displayed in the title bar
- initially invisible and remains so until `setVisible(true)` method is called
- Swing `JFrame` extends `java.awt.Frame` and has same basic behaviour
  - main difference is components are added to contentPane [retrieved via `getContentPane()`] not directly to `JFrame`
  - usually add a container (`JPanel`) to contentPane and then manage components in the `JPanel`

java.awt.Dialog

- `java.awt.Dialog`
  - similar to frame (provides a border and title and is resizable)
  - however it cannot have a menu or be iconized
  - must be anchored to a frame (same as Window)
- Dialogs can be modal => when `setVisble(true)` is called:
  - input to the Dialog's ancestors (usually a `Frame`) is blocked
  - the thread that launched the dialog is suspended until the dialog is closed
  - non-modal behaviour is default if not specified with constructor
- AWT provides limited Dialog support
  - `FileDialog` is the only custom dialog
  - message, yes/no and question dialogs must be hand coded
  - Swing provides JOptionPane class with static methods for creating basic dialogs
  - e.g. `JOptionPane.showMessageDialog(...)`
- Relationship between Swing `JDialog` and AWT `Dialog` is same as `Frame/JFrame`

Sizing and Moving Windows

- All windows (i.e. frames and dialogs) inherit from `Component` => use the methods of component class for sizing etc.

```
void setLocation(int x, int y)
void setSize(int width, int height)
void setBounds(int x, int y, int width, int height)
e.g.      Frame f=new Frame("Test");
          f.setSize(100,100);
          f.setLocation(0,0);
          f.show();
```

- Usually only size top level containers (Frame, Applet etc.) otherwise use layout management for sizing

Example: JMenuBarDemo (Pull down menus)

For additional help using menus see
http://java.sun.com/docs/books/tutorial/uiswing/components/menu.html

Also look at **JPopupMenu** (which is not covered explicitly in this lecture)

In this example we will look at incorporating a menu bar into a GUI application.

New concepts covered in this example include:

- Adding a Swing JMenuBar to a JFrame
- Creating and manipulating JMenuItem components.
- Using a FileChooser to interact with the underlying file system
- Using a DialogBox to display output to the user
- Adding a JScrollPane to a container

We will also explore a different way of setting up event Listeners for components in your GUI – namely using (anonymous) inner classes.

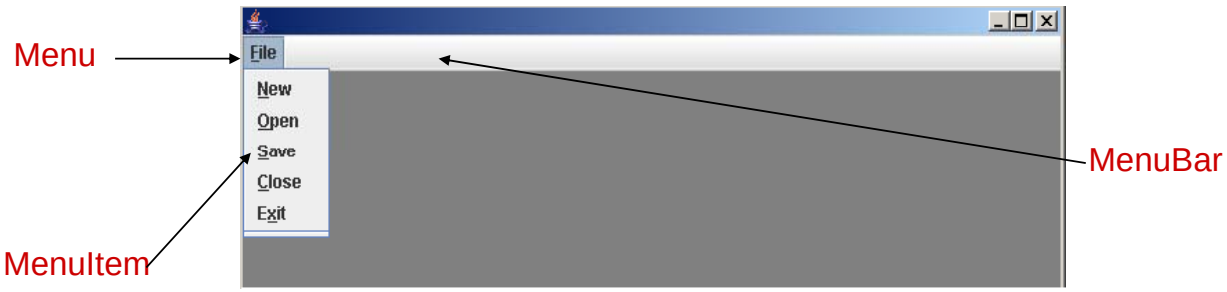
Adding a Menu Bar to a JFrame

A Menu bar consists of three different parts – the J/MenuBar component itself, one or more J/Menu components (which are added to the J/MenuBar) and a set of J/MenuItem components for each Menu component which represent the menu options.

To add a menu bar to a JFrame as shown in the example below you need to create the J/MenuBar, create one or more Menu components, fill each of the Menu components with their corresponding J/MenuItem components and add each of the (now filled) Menu components to the J/MenuBar.

Once the MenuBar has been set up it can be added to the JFrame using the method:

void setJMenuBar(JMenuBar menubar);



Example: Setting up a JMenuBar for a JFrame

To set up a menu bar you need to create the JMenuBar component, create one or more Menu components, create and add the required MenuItem components to the Menu Objects and then add each of the Menu objects to the JMenuBar, after which the JMenuBar can be added to your JFrame.

```
import java.awt.*; import java.awt.event.*; import java.io.*;
import javax.swing.*; import javax.swing.filechooser.*;

class JMenuBarDemo extends JFrame {
   private JTextArea text; private JScrollPane scroll;
   private JMenuBar menu;  private JMenu fileMenu;

   private JMenuItem newItem;  private JMenuItem openItem;
   private JMenuItem saveItem; private JMenuItem closeItem;
   private JMenuItem exitItem; private JMenuItem lastFileItem;

   private final JFileChooser fileSelection;
   private Container contentPane;
   private boolean fileOpen;
```

JMenuBarDemo Example: Setting up the menu components

```
public JMenuBarDemo()
{
   fileOpen = false; menu = new JMenuBar();
   fileMenu = new JMenu("File");

   fileMenu.setMnemonic(KeyEvent.VK_F);
   menu.add(fileMenu);

   newItem   = new JMenuItem("New", KeyEvent.VK_N);
   openItem  = new JMenuItem("Open", KeyEvent.VK_O);
   saveItem  = new JMenuItem("Save", KeyEvent.VK_S);
   closeItem = new JMenuItem("Close", KeyEvent.VK_C);
   exitItem  = new JMenuItem("Exit", KeyEvent.VK_X);


   fileSelection = new JFileChooser(new File("H:\\"));
   text = new JTextArea(20,50);
   text.setFont(new Font("Courier New",Font.PLAIN,12));

   scroll = new JScrollPane(text);
```

This is what is called a "mnemonic" - it is basically a shortcut key for the menu (this shortcut is "Alt-F").

Menu items can also have mnemonics

Create a file selection dialog for later use

Add a scroll pane to the TextArea component

JMenuBarDemo example: Adding a listener using an inner class

An (anonymous) inner class if effectively a one-use class that is directly associated with another class (or object). In an event-driven program it is a quick and easy way to add the event handling (listener) code directly to a specific component – thus there is no need to identify individual components in the listener code later on.

```
newItem.addActionListener(
              new ActionListener(){

public void actionPerformed(ActionEvent e) {

      if (!fileOpen) {
         fileOpen = true;
         contentPane.add(scroll);
         validate();
         contentPane.repaint();
      }
      else {
         text.setText("");
      }
   }});
```

Inner class definition

Listener for the "New" menu item

When we add components to a container at runtime we need to call the validate() method to make sure everything is laid out correctly

We also need to repaint the content pane to ensure that the background of the text area is "repainted"

JMenuBarDemo example - Using a JFileChooser / dialog

A file selection dialog is a popup window that you can use to explore the local fiile system and select files for use within your application. At the beginning of this example we created a new JFileChooser object as shown below – we can use this JFileChooser object as a selector for both "save" files and files we want to open.

```
fileSelection = new JFileChooser(new File("H:\\"));
```

We passed a new File object to the constructor above which represents the initial path the JFileChooser will start from. Now lets have a look at the code for the "open" listener.

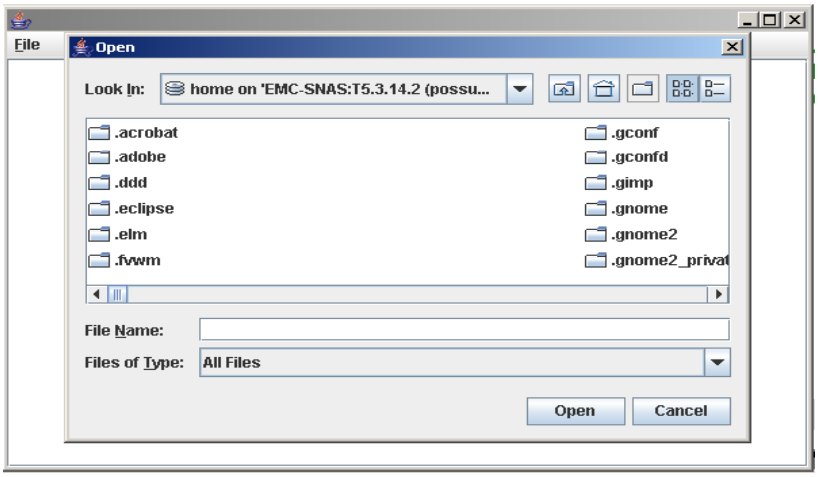
```
openItem.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent e) {
      int result;
      result = fileSelection.showOpenDialog(
                              JMenuBarDemo.this);

      if (result == JFileChooser.APPROVE_OPTION) {
         File f = fileSelection.getSelectedFile();
         JOptionPane.showMessageDialog(JMenuBarDemo.this,
              "Open file selected was " + f.getPath());
      }
   }});
```

Open file selection dialog

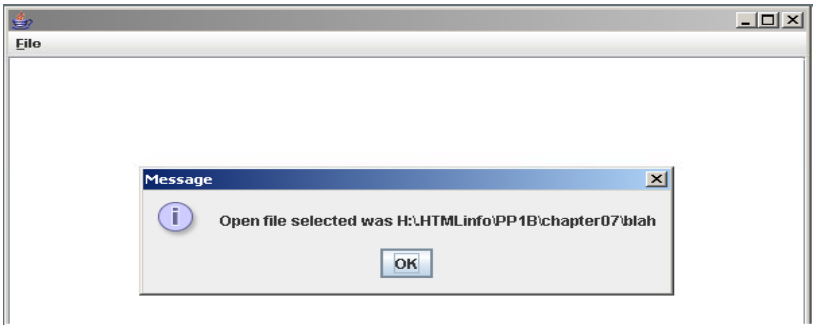
Process file if user clicks on "Open" button

JMenuBarDemo - Using a file selection dialog (cont)

The File Selection Dialog



The Dialog box



JMenuBarDemo example (cont)

```
saveItem.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent e) {
      int result;

      result = fileSelection.showSaveDialog(
                              JMenuBarDemo.this);

      if (result == JFileChooser.APPROVE_OPTION) {
         File f = fileSelection.getSelectedFile();
         JOptionPane.showMessageDialog(JMenuBarDemo.this,
              "Save file selected was " + f.getPath());
      }
   }});

 closeItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e){
       fileOpen = false; text.setText("");
       contentPane.remove(scroll);
       contentPane.repaint();
}});
```

Registering listeners to the save/close buttons

JMenuBarDemo example (cont)

```
      exitItem.addActionListener(new ActionListener(){
         public void actionPerformed(ActionEvent e) {
            System.exit(0);
         }});


      fileMenu.add(newItem);  fileMenu.add(openItem);
      fileMenu.add(saveItem); fileMenu.add(closeItem);
      fileMenu.add(exitItem); fileMenu.addSeparator();

      menu.add(fileMenu);  this.setJMenuBar(menu);

      contentPane = this.getContentPane();
      contentPane.setBackground(Color.gray);

      this.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            System.exit(0);
         }});
   }// end constructor
```

JMenuBarDemo example (cont)

```
   public static void main (String [] args) {
      JMenuBarDemo demo = new JMenuBarDemo();

      demo.setSize(600,400);
      demo.setLocation(250,250);
      demo.setVisible(true);
   }
} // end class
```